

Programmation 2 : deuxième cours

Arnaud Labourel arnaud.labourel@univ-amu.fr

17 ou 19 septembre 2018



Classes utiles et types primitifs

Classe Object

Par défaut, les classes étendent la classe `Object` de Java et ont donc les méthodes suivantes que l'on peut redéfinir :

- `boolean equals(Object obj)`: Indicates whether some other object is "equal to" this one.
- `String toString()`: Returns a string representation of the object.
- ...

Test d'égalité

- `o1 == o2` : vrai si `o1` et `o2` sont le même objet et faux sinon
- `o1.equals(o2)` : vrai si `o1` et `o2` correspondent à deux objets considérés égaux (exemple : étudiant ayant le même id, chaîne de caractères ayant les mêmes caractères, ...)

Les types primitifs

En java, il existe des types **primitifs** qui ne sont pas des objets :

type	catégorie	taille	valeurs possibles	affichage
byte	entier	8 bits	-128 à 127	0
short	entier	16 bits	-32768 à 32767	0
int	entier	32 bits	-2^{31} à $2^{31} - 1$	0
long	entier	64 bits	-2^{63} à $2^{63} - 1$	0
float	flottant	32 bits		0.0
double	flottant	64 bits		0.0
char	caractère	16 bits	caractère unicode	'\000'
boolean	booléen	non définie	false ou true	false

Comportement types primitifs

Lors d'un appel de méthode les arguments sont passés par valeur : une copie de la valeur de l'argument est créé lors de l'appel.

Pour les objets, cela signifie passer une copie de la référence : il est donc possible de modifier l'état de l'objet.

Pour les types primitifs, cela signifie que l'argument est un copie uniquement créée pour l'appel et toute modification de sa valeur n'aura pas d'impact en dehors de l'appel.

Tableaux unidimensionnels

En Java, les tableaux sont des objets (et donc des références).

Déclaration d'une variable de type "référence vers un tableau" :

```
int[] arrayOfInt;  
double[] arrayOfDouble;
```

Construction d'un tableau :

```
arrayOfInt = new int[10]  
arrayOfDouble = new double[3];
```

Utilisation d'un tableau :

```
arrayOfInt[0] = 5;  
arrayOfInt[9] = 10;  
arrayOfDouble[2] = arrayOfInt[0] / arrayOfInt[9];  
system.out.println(arrayOfDouble.length) // 3
```

Tableaux multidimensionnels

Déclaration :

```
int[] [] matrixOfInt;
```

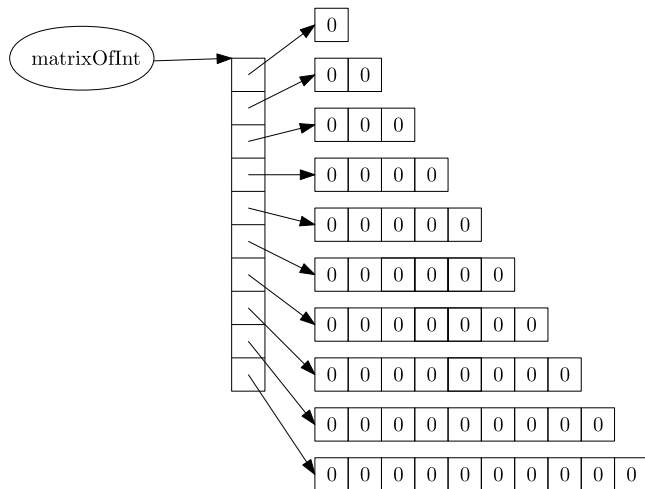
Construction :

```
matrixOfInt = new int[10] [];  
for(int row = 0; row < matrixOfInt.length; row++)  
    matrixOfInt[row] = new int[5];  
/* ou directement */  
matrix = new int[10][5];
```

Que produit le code suivant (réponse au transparent suivant) ?

```
matrixOfInt = new int[10] [];  
for(int row = 0; row < matrixOfInt.length; row++)  
    matrixOfInt[row] = new int[row + 1];
```

Tableau de tableaux



Chaînes de caractères (1/2)

La classe `String` permet de définir des chaînes de caractères invariables (**immutable**)

Déclaration et création :

```
String hello = "Hello";  
String world = "World";
```

Concaténation :

```
String helloWorld = hello + " " + world + " ! ";  
int integer = 13;  
String helloWorld1213 = hello + " " + world + " "  
                        + 12 + " " + integer;
```

Chaînes de caractères (2/2)

Affichage :

```
System.out.print(helloWorld); // affiche "Hello World"  
System.out.println(helloWorld); // affiche "Hello World"  
// avec retour à la ligne
```

Comparaison :

```
String a1 = "a";  
String a2 = "a";  
String a3 = new String("a");  
System.out.println(a1==a2); // affiche "true"  
System.out.println(a1==a3); // affiche "false"  
System.out.println(a1.equals(a3)); // affiche "true"
```

Les classes `ArrayList` et `LinkedList` : permettent de créer des listes en java.

```
List<String> strings = new ArrayList<>();  
strings.add("first");  
strings.add("second");  
System.out.println(strings);  
// affiche "[first, second]"
```

Structures de contrôles

La structure conditionnelle : if

La **structure conditionnelle** permet d'exécuter un bloc d'instructions que si une condition est réunie.

```
if (people.isGoodGuy()) {  
    Sout.println("Greetings, " + people.getName() + "!");  
}
```

```
if (people.isGoodGuy()) {  
    Sout.println("Greetings, " + people.getName() + "!");  
} else {  
    Sout.println("You are not welcome here, "  
                + people.getName() + ".");  
}
```

La répétition "tant que" : while(condition)

Le while permet de répéter un bloc d'instructions tant que sa condition est vraie :

```
int count = 10;
while (count > 0) {
    System.out.println(count + "...");
    count = count - 1;
}
System.out.println("BOUM!!!");
```

Attention

Le test n'est effectué que juste avant d'effectuer la première instruction du bloc à chaque itération !

La répétition pour toujours (*forever*) : `for(;;)`

La boucle `for(;;)` permet de répéter un bloc d'instructions tant qu'on ne quitte pas la boucle avec les instructions `break` ou `return`.

```
int count = 10;
for(;;) {
    System.out.println(count + "...");
    if (count <= 0) break;
    count = count - 1;
}
System.out.println("BOUM!!!");
```

Note

on peut avoir plusieurs instructions `break` dans la même boucle.

La répétition bornée : for

Dans certains cas, il est plus direct d'utiliser une boucle for, qui isole :

- l'initialisation de la boucle,
- la condition d'arrêt,
- l'instruction de progression.

```
for (int count = 10; count > 0; count = count - 1) {  
    System.out.println(count + "...");  
}  
System.out.println("BOUM!!!");
```

On peut aussi utiliser break et return dans une boucle for.

L'itération pour chaque : for

Pour les objets qui sont des **collections**, implémentant l'interface `Iterable<Elt>`, on peut utiliser la boucle `for` ainsi:

```
for (Elt item : myCollection) {  
    item.doSomething();  
    // ...  
}
```

Exemple de collections

- `List<Elt>`
- les tableaux

Les structures de contrôles définissent des blocs d'instructions, entre { et } :

- possible d'utiliser d'autres structures de contrôle dans ces blocs, par **imbrication**,
- les blocs définissent les **portées** des variables, une variable ne vit que dans le bloc où elle est définie et les blocs imbriqués dedans.
- les structures de contrôle compliquent la lecture du code par un humain : il faut éviter l'**imbrication** !

Tests et développement

Règle

Un code non testé n'a aucune valeur.

Corollaire

Tout code doit être testé

Différents type de tests

- **Test unitaires** : Tester les différentes parties d'un programme indépendamment les unes des autres.
- **Test de non régression** : Vérifier que le nouveau code ajouté ne corrompt pas les codes précédents : les tests précédemment réussis doivent encore l'être.

Tests unitaires

- Tester une unité de code : classe, méthodes, ...
- Vérifier un comportement :
 - ▶ cas normaux
 - ▶ cas limites
 - ▶ cas anormaux

Tests unitaires en java : JUnit

- Un framework de test unitaire pour Java
- S'appuie sur des **assertions**

Assertions JUnit (1/2)

- `assertTrue(boolean condition)` : vérifie que condition est vraie.
- `assertFalse(boolean condition)` : vérifie que condition est faux.
- `assertEquals(expected, actual)` : vérifie que expected est égal à actual
égal : equals pour les objets et == pour les types primitifs.
- `assertEquals(double expected, double actual, double delta)` : vérifie que $|expected - actual| \leq delta$
- `assertNull(Object object)` : vérifie que la référence est null
- `assertNotNull(Object object)` : vérifie que la référence **n'est pas** null

Assertions JUnit (2/2)

- `assertSame(Object expected, Object actual)` : vérifie que les deux objets sont les mêmes (même référence).
- `assertArrayEquals(Object[] expected, Object[] actual)` : vérifie si les deux tableaux contiennent les mêmes éléments dans le même ordre.
- `fail()` : échoue toujours

Message

Pour toutes les méthodes `assert`, il est possible de mettre un message en premier paramètre qui permet d'identifier l'assertion.

Principe

- Le code d'un projet est stocké dans un serveur.
- Les développeurs soumettent des modifications avec des commentaires à chaque fois.
- Le serveur conserve l'historique des mises à jour

Pourquoi la gestion de version ?

- Pour travailler de manière harmonieuse en équipe sans se marcher dessus
- Pour revenir en arrière en cas de problèmes
- Possibilité de faire valider le code (via des tests) par le serveur et de rendre le déploiement automatique

- Logiciel de gestion de version le plus populaire
- Serveur gratuit : github
- Version libre de logiciel serveur : gitlab
- Gestion de version décentralisée : la gestion de version se fait aussi en local

Utilisation de git

- Via l'IDE : VCS (Version Control Systems) dans le menu d'IntelliJ
- En ligne de commande : commande git

Exemple de commandes git

```
git clone adresse_projet
```

⇒ Clone un projet en local depuis un serveur

```
git add nom_de_fichier
```

⇒ Ajoute un fichier à la prochaine mise à jour.

```
git commit -m"commentaire"
```

⇒ Fait une mise à jour en local

```
git push
```

⇒ Pousse les mises à jour locales sur le serveur

```
git pull
```

⇒ Récupère les mises à jour du serveur en local

Types paramétrés

Stack d'Object

Supposons que nous ayons la classe suivante :

```
public class Stack {
    private Object[] stack = new Object[100];
    private int size = 0;
    public void push(Object object) {
        stack[size] = object; size++;
    }

    public Object pop() {
        size--;
        Object object = stack[size];
        stack[size]=null; // Pour le Garbage Collector.
        return object;
    }
}
```

Problème de Stack d'Object

Nous rencontrons le problème suivant :

```
Stack stack = new Stack();  
String string = "truc";  
stack.push(string);  
string = (String)stack.pop();  
// Transtypage obligatoire !
```

Nous avons également le problème suivant :

```
Stack stack = new Stack();  
Integer intValue = new Integer(2);  
stack.push(intValue);  
String string = (String)stack.pop();  
// Erreur à l'exécution
```

La solution : types paramétrés

Par conséquent, on souhaiterait pouvoir préciser le type des éléments :

```
Stack<String> stack = new Stack<String>();  
String string = "truc";  
stack.push(string); // Le paramètre doit être un String.  
String string = stack.pop(); // retourne un String.
```

Java nous permet de définir une classe Stack qui prend en paramètre un type. Ce type paramétré va pouvoir être utilisé dans les signatures des méthodes et lors de la définition des champs de la classe.

Lors de la compilation, Java va utiliser le type paramétré pour effectuer :

- des vérifications de type ;
- des transtypages automatiques ;
- des opérations d'emballage ou de déballage de valeurs.

Définition de classes paramétrées

La nouvelle version de la classe Stack :

```
public class Stack<T> {  
    private Object[] stack = new Object[100];  
    private int size = 0;  
    public void push(T element) {  
        stack[size] = element;  
        size++;  
    }  
    public T pop() {  
        size--;  
        T element = (T)stack[size];  
        stack[size] = null;  
        return element;  
    }  
}
```

Emballage et déballage

Les types primitifs ne sont pas des classes :

Dans le cas d'un `int`, on doit utiliser la classe d'emballage (wrapper class) `Integer` :

Interdit : ~~`Stack<int> stack = new Stack<int>();`~~

Autorisé :

```
Stack<Integer> stack = new Stack<Integer>();
int intValue = 2;
Integer integer = new Integer(intValue);
// → emballage du int dans un Integer.
stack.push(integer);
Integer otherInteger = stack.pop();
int otherIntValue = otherInteger.intValue();
// → déballage du int présent dans le Integer.
```


Types primitifs

type	classe d'emballage	taille	valeurs possibles
byte	Byte	8 bits	-128 à 127
short	Short	16 bits	-32768 à 32767
int	Integer	32 bits	-2^{31} à $2^{31} - 1$
long	Long	64 bits	-2^{63} à $2^{63} - 1$
float	Float	32 bits	
double	Double	64 bits	
char	Character	16 bits	caractère unicode
boolean	Boolean	non définie	false ou true

Classes d'emballage

La classe `Number` sert de base pour toutes les classes d'emballage.

Elle contient les méthodes suivantes :

- `public int intValue()`
- `public long longValue()`
- `public float floatValue()`
- `public double doubleValue()`
- `public byte byteValue()`
- `public short shortValue()`

Les classes d'emballage étendent `Number` :

- `Byte` → `public static Byte valueOf(byte b)`
- `Short` → `public static Short valueOf(short s)`
- `Integer` → `public static Integer valueOf(int i)`
- `Long` → `public static Long valueOf(long l)`
- `Byte` → `public static Byte valueOf(byte b)`

Ils existent des constructeurs mais ils sont dépréciés (et donc pas à utiliser).

La classe Character

Les classes d'emballage ne contiennent pas que des méthodes liées aux instances :

- `public static Byte valueOf(byte b)`
- `public static char charValue()`
- `public static boolean isLowerCase(char ch)`
- `public static boolean isUpperCase(char ch)`
- `public static boolean isDigit(char ch)`
- `public static boolean isLetter(char ch)`
- `public static boolean isLetterOrDigit(char ch)`
- `public static char toLowerCase(char ch)`
- `public static char toUpperCase(char ch)`
- `public static char toTitleCase(char ch)`

Emballage et déballage automatique

Depuis Java 5, il existe l'emballage et le déballage automatique :

```
Stack<Integer> stack = new Stack<Integer>();  
int intValue = 2;  
stack.push(intValue);  
// → emballage automatique du int dans un Integer.  
int otherIntValue = stack.pop();  
// → déballage automatique du int.
```

Attention

Il est important de noter que des allocations sont effectuées lors des emballages sans que des `new` soient présents dans le code.

Exemple : liste chaînée générique

On considère une liste chaînée de String

```
public class LinkedList {
    private class Node {
        private String data;
        private Node next;
        public Node(String data, Node next) {
            this.data = data;
            this.next = next;
        }
    }
    private Node first = null;
    public void add(String data) {
        first = new Node(data, first);
    }
}
```

Exemple : liste chaînée générique

Nous la transformons en classe paramétrée de la façon suivante :

```
public class LinkedList<T> {
    private class Node {
        private T data;
        private Node next;
        public Node(T data, Node next) {
            this.data = data;
            this.next = next;
        }
    }
    private Node first = null;
    public void add(T data) {
        first = new Node(data, first);
    }
}
```

Plusieurs paramètres de types

```
public class Pair<A, B> {  
    public A first;  
    public B second;  
    public Pair(A first, B second) {  
        this.first = first;  
        this.second = second;  
    }  
    public static <A, B> Pair<A,B>  
        makePair(A first, B second) {  
        return new Pair<A,B>(first, second);  
    }  
}
```


Utilisation d'une classe avec plusieurs paramètres de types

```
public class Main {  
  
    public static void main(String[] args) {  
        Pair<String,Integer> pair =  
            Pair.<String,Integer>makePair("tot",12);  
        System.out.println(pair);  
    }  
}
```